Web Search Engine Report

CS6913

Zijie Zhu (zz1613)

1. Introduction

This simple web search engine implement functions:

➢ Conjunctive Search

User input a search query consisting of words split with space, and the program matches the document containing all words appeared in the query, returns the top 10 search results with the content {doc id, url, rank score and snippet}

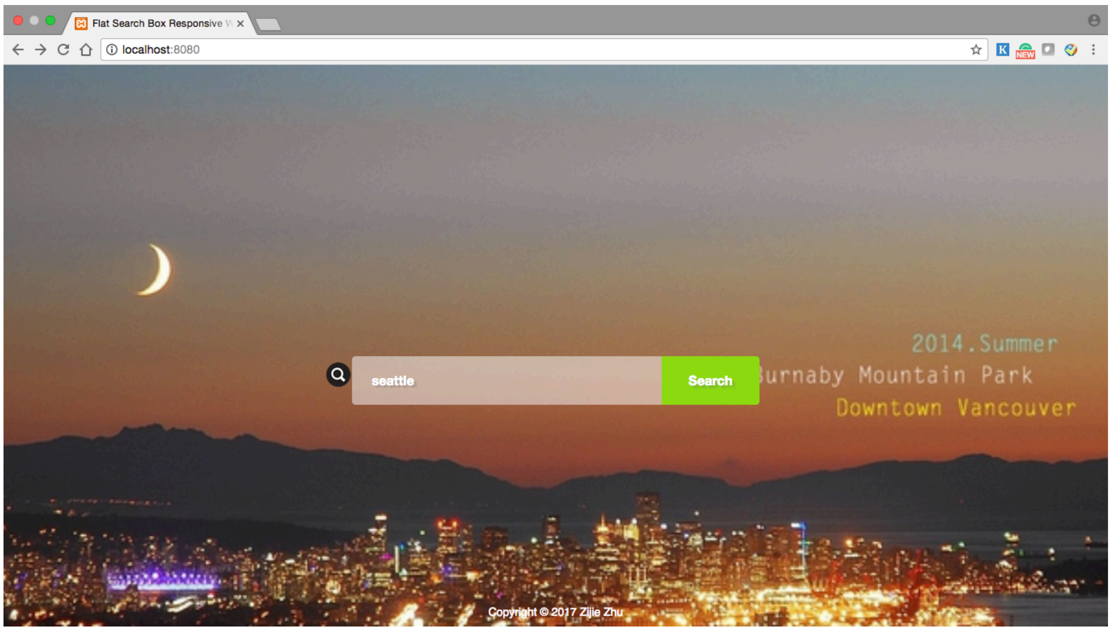
➢ Disjunctive Search

User input a search query consisting of words split with space, and the program matches the document containing any word appeared in the query,

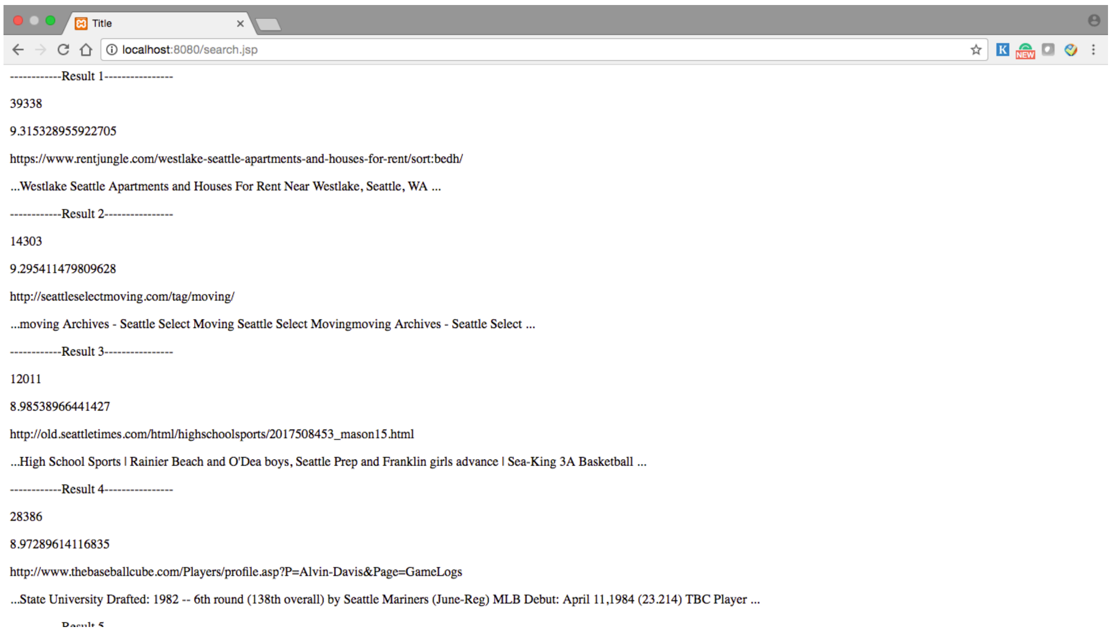
returns the top 10 search results with the content {doc id, url, rank score and snippet}

Program Screenshot

➢ Search Home

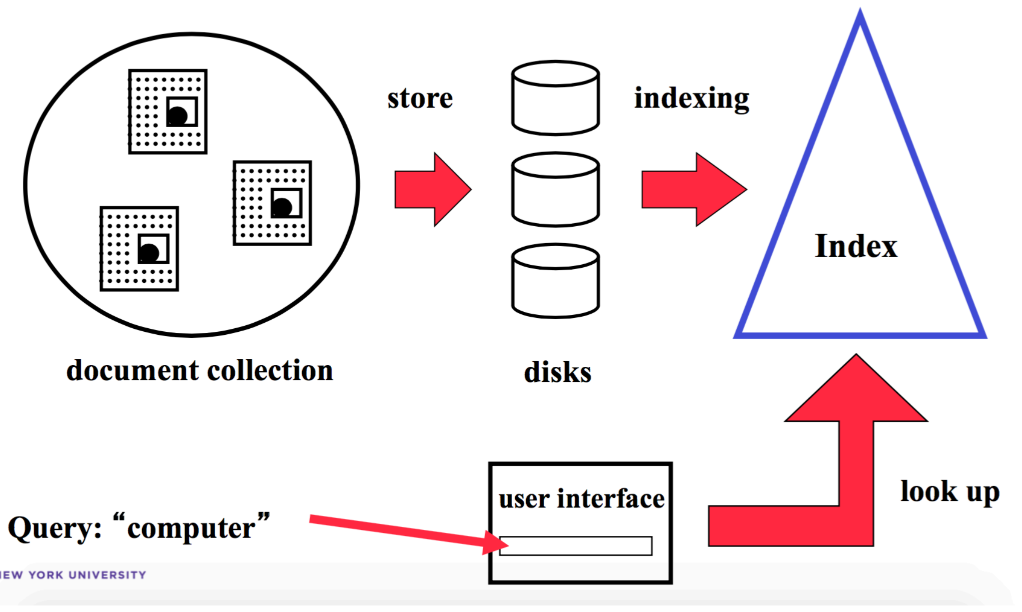


➢ Search Result



```
-----------Result 1---------------

39338

9.315328955922705

https://www.rentjungle.com/westlake-seattle-apartments-and-houses-for-rent/sort:bedh/

...Westlake Seattle Apartments and Houses For Rent Near Westlake, Seattle, WA ...

-----------Result 2---------------

14303

9.295411479809628

http://seattleselectmoving.com/tag/moving/

...moving Archives - Seattle Select Moving Seattle Select Movingmoving Archives - Seattle Select ...

-----------Result 3---------------

12011

8.98538966441427

http://old.seattletimes.com/html/highschoolsports/2017508453_mason15.html

...High School Sports | Rainier Beach and O'Dea boys, Seattle Prep and Franklin girls advance | Sea-King 3A Basketball ...

-----------Result 4---------------

28386

8.97289614116835

http://www.thebaseballcube.com/Players/profile.asp?P=Alvin-Davis&Page=GameLogs

...State University Drafted: 1982 -- 6th round (138th overall) by Seattle Mariners (June-Reg) MLB Debut: April 11,1984 (23.214) TBC Player ...
```

2. Architecture



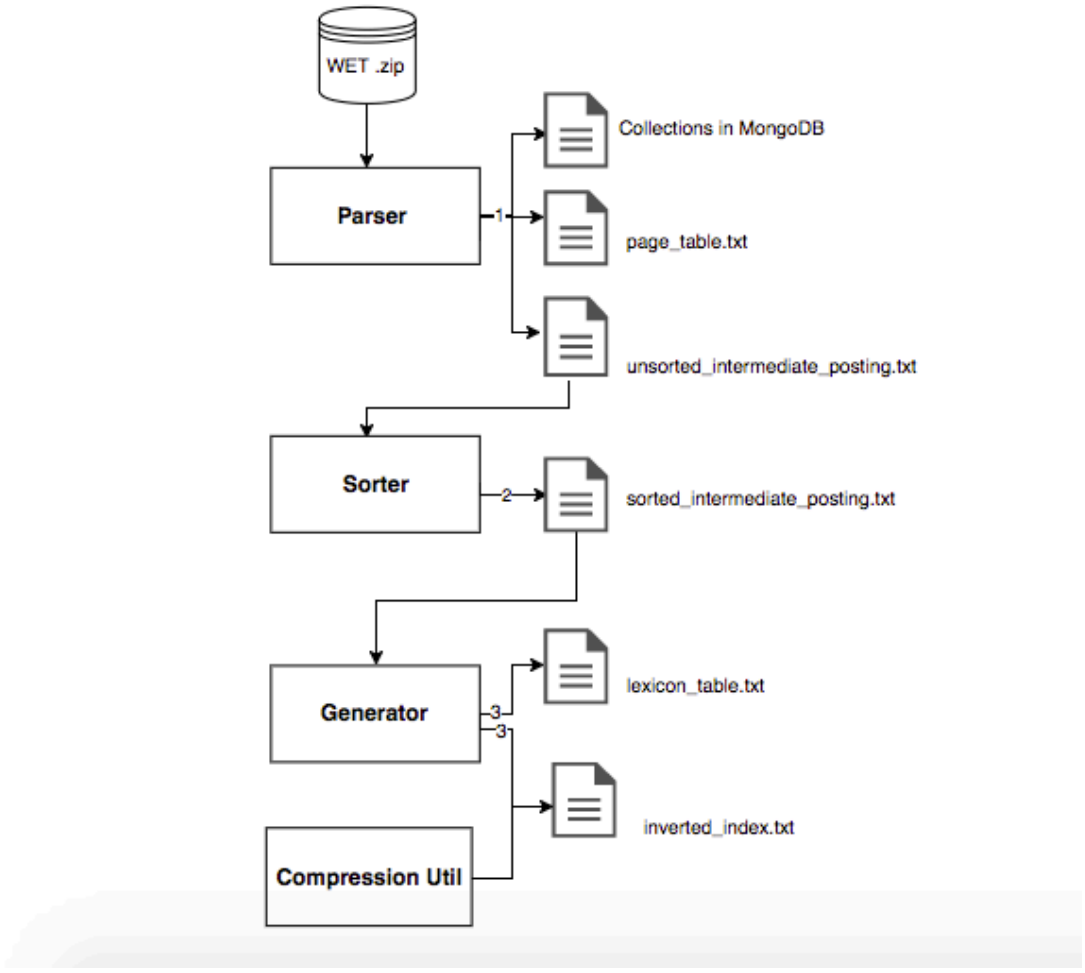
In the next chapter, I will explain the store, indexing and look up part from design and implementation perspective. The store and indexing part will be covered in Inverted Index Generation module, and the look up part will be covered in Query module.

3. Module Design & Implementation

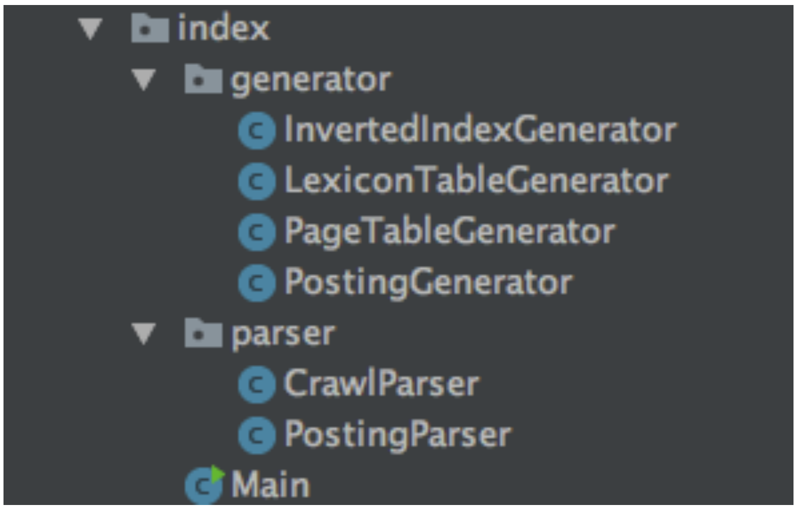
3.1 Indexing

3.1.1 Overview

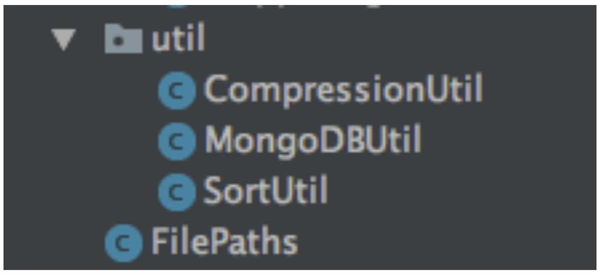
➢ **High–level Overview**



➢ **Lower-level Overview**



```
index
    generator
        InvertedIndexGenerator
        LexiconTableGenerator
        PageTableGenerator
        PostingGenerator
    parser
        CrawlParser
        PostingParser
    Main
```



The main components in indexing module are crawler, sorter and generator, which work sequentially. In the next, we will explain each component in detail.

3.1.2 Parser: generate intermediate postings and page table

There should be a crawler which crawl the web page contents online before the parser, instead, we use existing crawling data by Common Crawl, an organizat –
ion provides large web crawls to the public and researchers.

Module Input

The input for parser module is raw crawl data by Common Crawl, which are stored in the WET file format. Thus we download the WET file under project dictionary, and use them after loading and compressing.

Main Functions & Output

- Generate page table: page_table.txt

  Get the page size of each page while parsing, and store them sequentially (in the order of docID) to disk with the following entry format:

Page Table.txt	
docID (int)	page size (int)

  Note: The reason we keep page url in mongoDB instead of page table is main memory can' hold lexicon table and page table containing page urls at the same time.

- Generate unsorted intermediate posting: unsorted_intermediate_posting.txt

  Get the words and count their frequency in each page while parsing, and store them (partially sorted in the order of docID) to disk with the following entry format:

Unsorted Intermediate Posting.txt		
docID (int)	word (string)	freq (int)

Note: We only consider English words, while getting the words in each page. And we didn't focus on the accuracy of the parsed words in the pages.

➢ Store original page and url to Mongo DB: data in mongoDB collections

1. Get the whole raw page contents of each page and store them to mongoDB, an document – oriented database, for later use of snippet generation.

Page Content (the collection name of MongoDB)	
url (string)	page content (string)

2. Get the page url of each page and store them to mongoDB for later use of search result generation.

Page Url (the collection name of MongoDB)	
docID (int)	url (string)

Tricky Part

➢ Main memory limitation

See beyond that main memory cannot hold lexicon table and page table containing page urls at the same time. So should store url separately to database.

➢ Create the index for mongoDB collections

MongoDB scan the whole collections sequentially to match the query by default, so it could be very slow when the match at the end of collection. Create the index for collections could solve this problem.

➢ Deal with large url

Creating index for Page Url collection in MongoDB failed because of some entries has too large url (over 1024 bytes) which cannot indexed on. So addressing large url during parsing.

3.1.3 Sorter: sort the intermediate postings

Module Input

Unsorted Intermediate Postings.txt from the Parser module.

Main Function & Output

➢ Sort the unsorted intermediate postings: sorted_intermediate_posting.txt

The entry of the output has following format:

Sorted Intermediate Posting.txt		
docID (int)	word (string)	freq (int)

Intermediate posting should be sorted by the word firstly and docID secondly to be used by generator later. We use Unix Sort to achieve sort by call bash commend from the program.

Unix sort bash cmd: sort -k1,1 -k2n,2 unsorted_intermediate_posting.txt > sorted_intermediate_posting.txt

3.1.4 Generator: generate inverted index and lexicon table

Module Input

Sorted Intermediate Postings.txt from the Sorter module.

Main Function & Output

➢ Generate inverted index: inverted_index.txt in binary format

1. The entry of inverted index has the following format:

Inverted Index	
block data	metadata

Block data part contains all docID and frequencies information consisting of multiple blocks, and the metadata contains information about every block.

We will show the interior structure of block data and metadata in the next.

Note: We don't keep the word itself in the entry as the lexicon table will store the position of the word’s metadata.

2. The block data has the following structure:

Block Data						
block 0 (128 postings)		block 1 (128 postings)		....	block n	
[docIDs]	[freqs]	[docIDs]	[freqs]	....	[docIDs]	[freqs]

Get the word, all docIDs along with the word frequencies of the pages containing this word during parsing the input file, and continually append to current block. Treat 128 postings (a posting means a pair of {docID, freq}) as an block, and write to disk every time the current block is full or finishing current word processing.

Inside a block, we store the docIDs and freqs in the format [docIDs], [freqs] instead of [{docID, freqs}] to apply calculate gap difference on [docIDs] which will save storage.

3. The metadata has the following structure:

Metadata			
metadata for block 0	metadata for block 1	...	metadata for block n
{max docId, start, size}	{max docId, start, size}	...	{max docId, start, size}

Metadata contains the metadata information for each block including:

Max docId: the max docId of the block

Start: the start position of a block

Size: the size of a block

We append a block metadata to Metadata after writing the block disk, and writing the Metadata to disk after finishing the blocks.

4. For compression, we apply gap difference on [docIds] in block, and compress both block data and metadata with var byte compression. Thus, Inverted Index file is in binary format. We will explain the compression technology in the next chapter.

➢ Generate lexicon table: lexicon_table.txt

Lexicon Table.txt			
word (String)	start (long)	size (int)	amount (int)

The entry of lexicon table has following format:

Start: the start position of the metadata part in inverted index of this word

Size: the size of the metadata part in inverted index of this word

Amount: the amount of the pages containing the word

3.1.5 Compression: Gap Difference + Var Byte

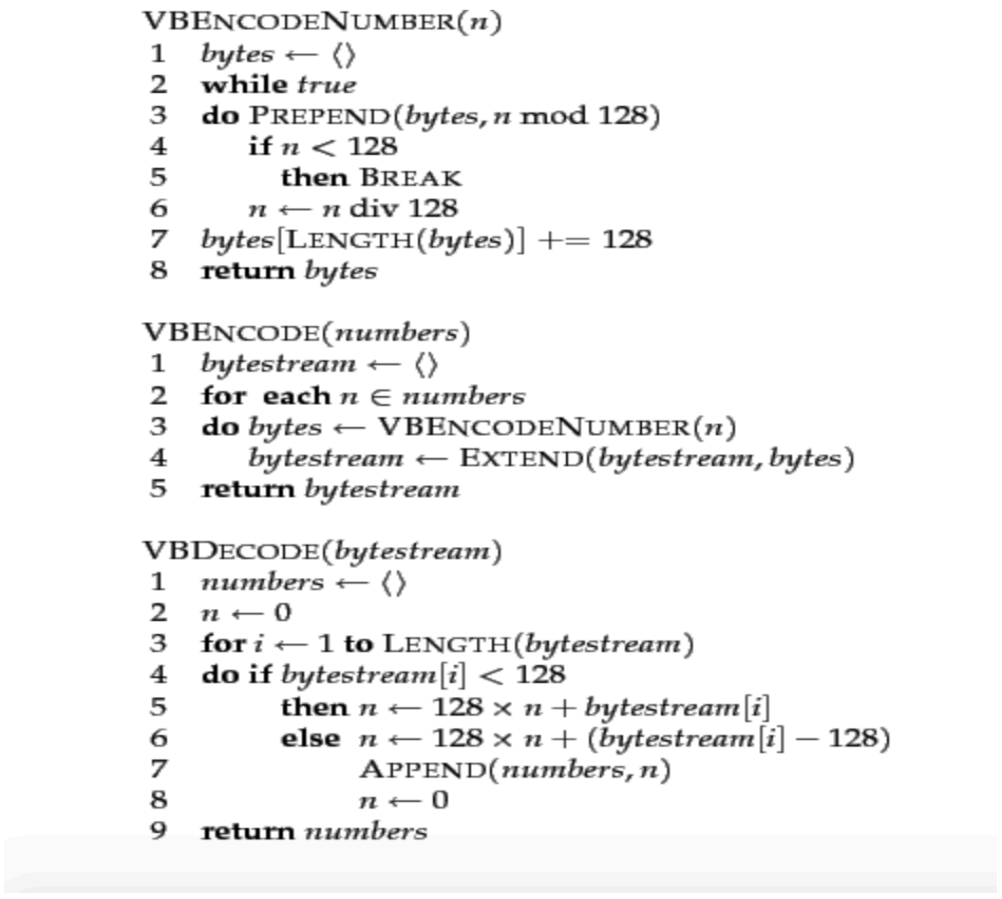
We apply gap difference on [docIDs] in block data part of inverted index, and then apply var byte compression for every entry in Inverted Index table before store them to disk. We will explain the idea of calculating gap difference and var byte compression in the next.

1. The idea of calculating gap difference:

As the docIDs is sorted, so from the end to the front, we use the gap value with previous value to represent the current value. In this way, all values will be much smaller than its original value.

2. The idea of var byte compression:

Use 7 bits in every 8 bit byte to store the integer and use the remain 1 bit to indicate whether this is the last byte that stores the integer or whether another byte follows. The pseudo code like following:



```
VBENCODENUMBER($n$)
bytes ← ⟨⟩
while true
do PREPEND(bytes, $n$ mod 128)
    if $n < 128$
        then BREAK
    $n \leftarrow n$ div 128
bytes[LENGTH(bytes)] += 128
return bytes

VBENCODE(numbers)
bytestream ← ⟨⟩
for each $n \in$ numbers
do bytes ← VBENCODENUMBER($n$)
    bytestream ← EXTEND(bytestream, bytes)
return bytestream

VBDECODE(bytestream)
numbers ← ⟨⟩
$n \leftarrow 0$
for $i \leftarrow 1$ to LENGTH(bytestream)
do if bytestream[$i$] $< 128$
        then $n \leftarrow 128 \times n +$ bytestream[$i$]
        else $n \leftarrow 128 \times n +$ (bytestream[$i$] $- 128$)
            APPEND(numbers, $n$)
            $n \leftarrow 0$
return numbers
```

Inverted index table is in binary format after compression and 2-3 times smaller than its original size

Tricky Part

- **Use long instead of int type for the position variable in inverted index**

  For 4 million pages, the position of inverted index entries could over the upper limit of integer. So see beyond this point could avoid a lot of refactor work after.

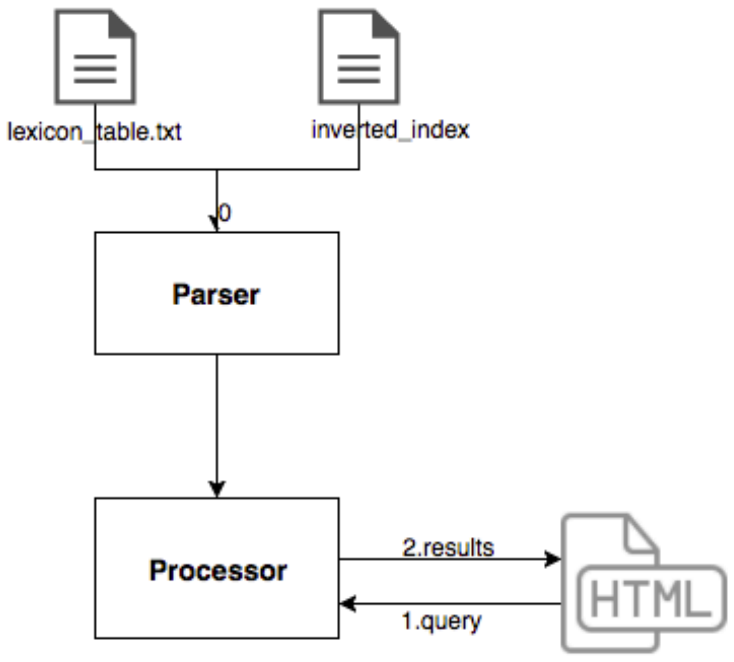
- **The var byte compression should be able to compress long type number.**

  As the position variable is long type, var byte compression should compress the long type number as well, which may requires a little bit modification for the algorithm.

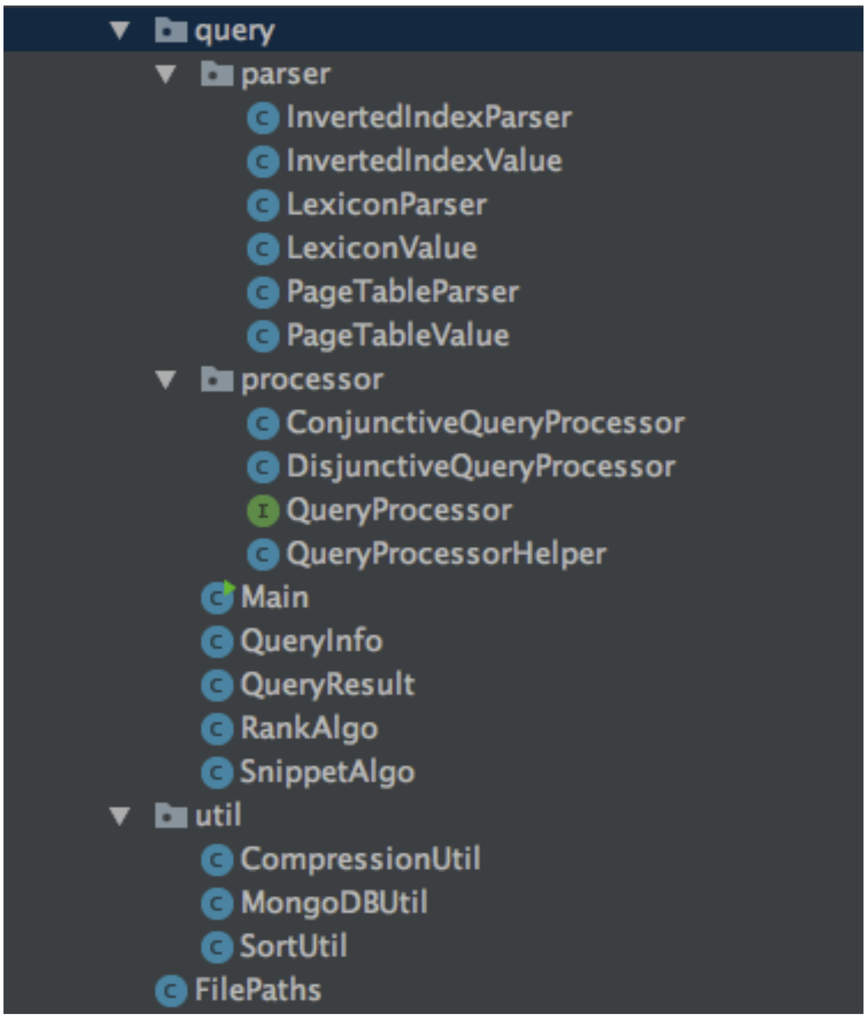
3.2 Query

3.2.1 Overview

➢ **High-level Overview**



➢ **Lower-level Overview**



```
▼ query
    ▼ parser
        InvertedIndexParser
        InvertedIndexValue
        LexiconParser
        LexiconValue
        PageTableParser
        PageTableValue
    ▼ processor
        ConjunctiveQueryProcessor
        DisjunctiveQueryProcessor
        QueryProcessor
        QueryProcessorHelper
      Main
      QueryInfo
      QueryResult
      RankAlgo
      SnippetAlgo
  ▼ util
      CompressionUtil
      MongoDBUtil
      SortUtil
    FilePaths
```

The main components in query module are parser and processor, which work sequentially. In the next, we will explain each component in detail.

3.2.2 Parser: Load tables to main memory

Module Input

Lexicon Table.txt and Page Table.txt from indexing module

Main Functions & Output

➢ Load Lexicon Table to main memory: Map<String, LexiconValue>

We read the Lexicon Table.txt from disk to main memory for later search use. Lexicon Table in main memory has the following structure:

Lexicon Table Map	
Key	Value
word (String)	{start, size, amount} (LexiconValue)

➢ Load Page Table to main memory: List<Integer>

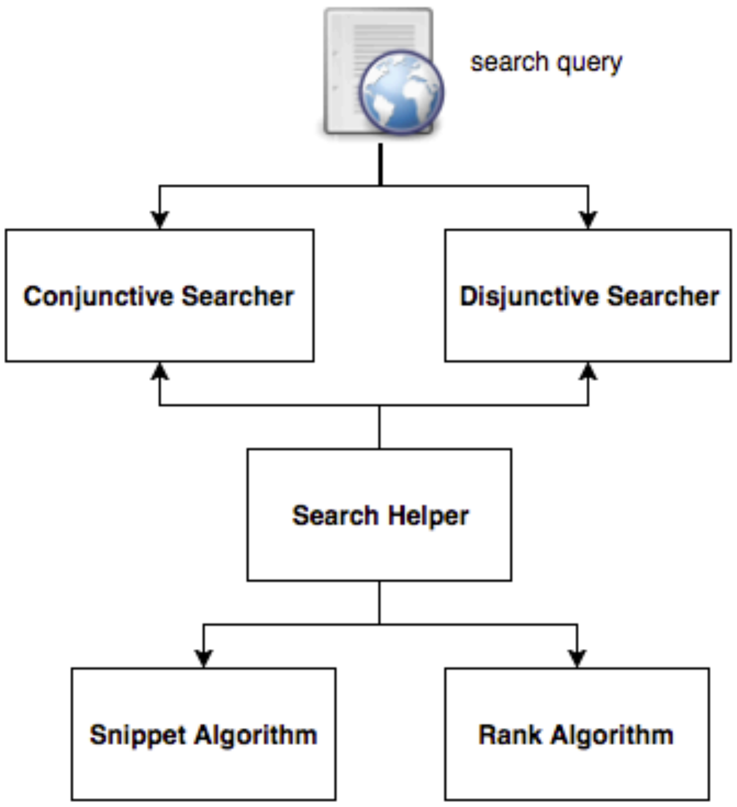
We read the Page Table.txt from disk to main memory for later search use.

Page Table.txt contains only docID along with the page size in the order of docID, so we store the page size to the list sequentially.

3.2.3 Processor: process search query

Processor module has more complicated structures, so we will start from an overview graph presenting the relationships between the sub-components, and explain them in detail in the next chapters.

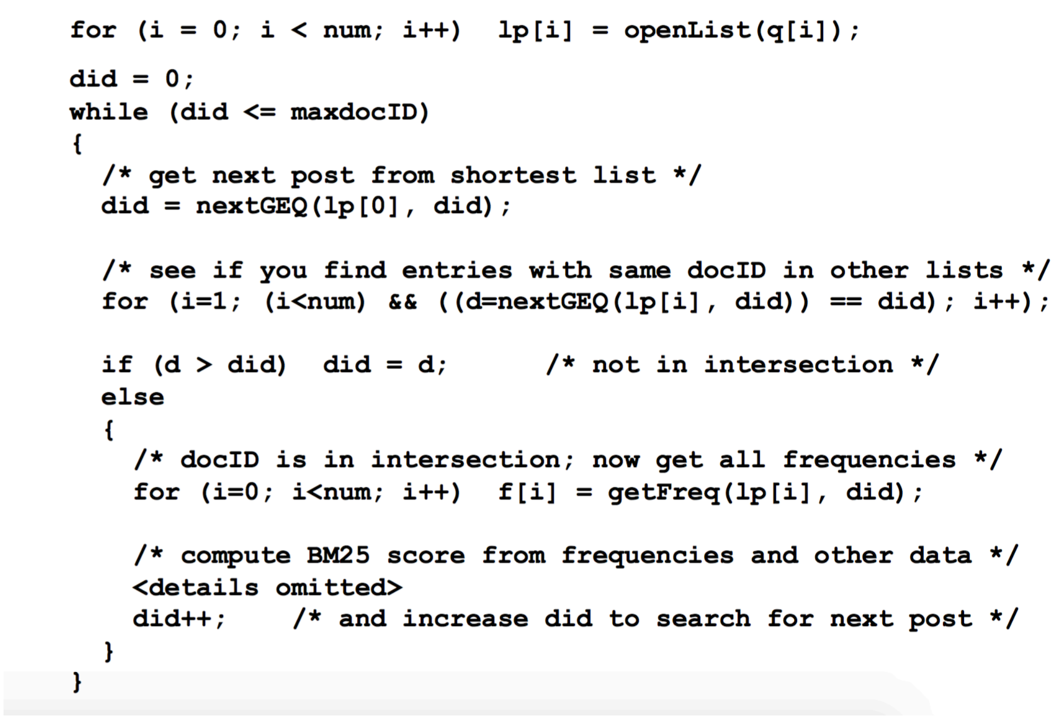
3.2.3.1 Overview



3.2.3.2 Conjunctive Searcher

Conjunctive searcher is responsible for processing conjunctive search, which only matches the pages containing all words in the query.

We use Document-AT-A-Time to implement the conjunctive search. The basic idea can be shown with the pseudo code:



```
for (i = 0; i < num; i++)  lp[i] = openList(q[i]);
did = 0;
while (did <= maxdocID)
{
  /* get next post from shortest list */
  did = nextGEQ(lp[0], did);

  /* see if you find entries with same docID in other lists */
  for (i=1; (i<num) && ((d=nextGEQ(lp[i], did)) == did); i++);

  if (d > did)  did = d;      /* not in intersection */
  else
  {
    /* docID is in intersection; now get all frequencies */
    for (i=0; i<num; i++)  f[i] = getFreq(lp[i], did);

    /* compute BM25 score from frequencies and other data */
    <details omitted>
    did++;    /* and increase did to search for next post */
  }
}
```

We process the words in the same time, suppose there are two words in the query, for docID in word1, we go to check whether word2 has this docID, if it has then output this docID along with the information to calculate rank and add to priority queue; if not, return current docID in word2 and repeat the process from the start.

3.2.3.3 Disjunctive Searcher

Disjunctive searcher is responsible for processing disjunctive search, which matches the pages containing any words in the query.

We use Term-AT-A-Time to implement disjunctive search, the basic idea is: process one inverted list at a time, for each posting in a list, create an hash entry with the docID as key and the cosine contribution for this term as value, for other lists, check if there is an entry in the hash table, if yes, add term contribution to cosine, if not, create a new entry results in the hash table.

Finally, add all entries of the hash map to the priority queue to get the top 10 results.

3.2.3.4 Search Helper: basic operations

Search Helper contains the basic operations shared between conjunctive searcher and disjunctive searcher including:

- openList (word)

  We do openList () for every word in the query

  We get the word's metadata information from Lexicon Table and then load the metadata part of this word in the inverted index file.

  The whole process including read and decode to store to a ArrayList in the main memory.

- nextGEQ (lp, k)

  We use nextGEQ(lp, k) to find the next posting in list lp with docID >= k and return its docID. Return -1 if none exists.

  The whole process is read the metadata of this word to get the block has maxDocID > k, then load this block to main memory as an ArrayList, and parse the block to find the minimum docID > k

- closeList ()

  We call closeList() after finishing an search to continue the next search process. Thus including clear all the containers with the data of the previous search process.

3.2.3.5 Rank Algorithm: BM25

Rank Algorithm is responsible for calculating the rank score for every page satisfies the query for later top 10 results selection.

We use BM25 as rank algorithm with the following formula:

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

$$K = k_1 \times \left((1-b) + b \times \frac{|d|}{|d|_{avg}}\right)$$

N: total number of documents in the collection

f_t: number of documents that contain term t

$f_{d,t}$: frequency of term t in document d

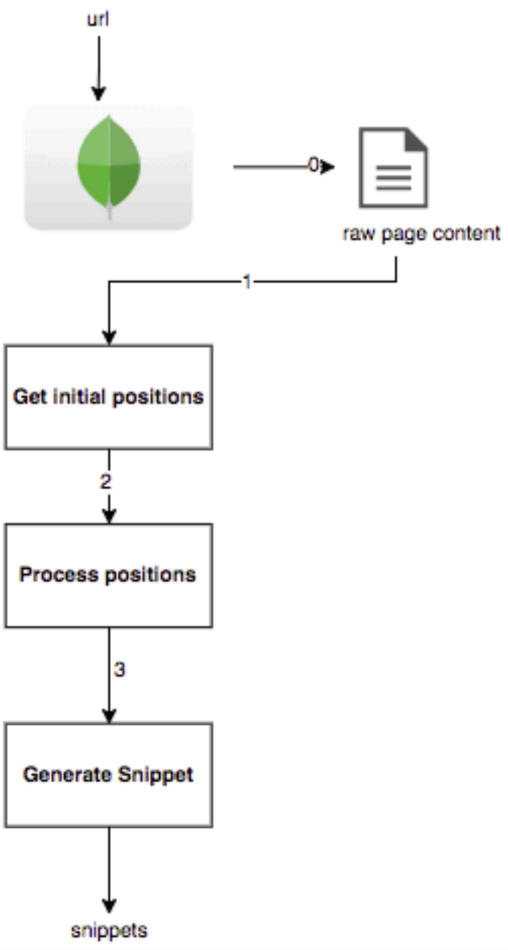
|d|: length of document d

$|d|_{avg}$: the average length of documents in the collection

k_1 and b: constants, usually $k_1 = 1.2$ and $b = 0.75$

3.2.3.6 Snippet Generation

Snippet generation is responsible for generating snippet containing all the words in the query (conjunctive search) or words appear in the matched page (disjunctive search). The overview of snippet generation could be shown with the following graph:



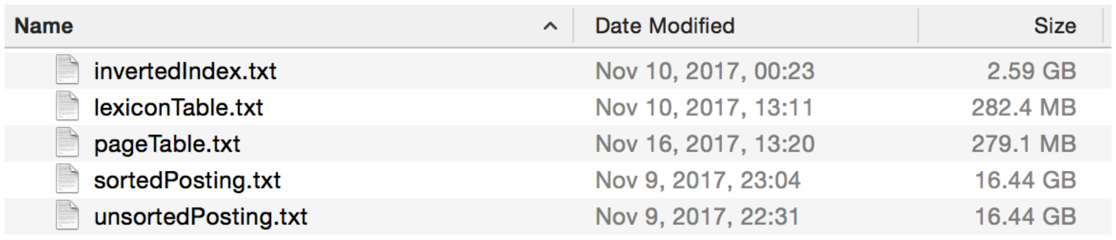
We do a little improvement for snippet algorithm in the step 3- process positions: Suppose there are three words in the conjunctive search query, and two of them are very close, if we just extract the line contains these two words, the result could contain duplicate part.

So the improvement is we determine the distance between sub-snippets, and do a intersection if needed.

4. Data Set Introduction

We use 80 wet zip which contains about 5 million pages as the source data.

The size of the outputs:



Name	Date Modified	Size
invertedIndex.txt	Nov 10, 2017, 00:23	2.59 GB
lexiconTable.txt	Nov 10, 2017, 13:11	282.4 MB
pageTable.txt	Nov 16, 2017, 13:20	279.1 MB
sortedPosting.txt	Nov 9, 2017, 23:04	16.44 GB
unsortedPosting.txt	Nov 9, 2017, 22:31	16.44 GB

5. Program Performance Evaluation

➢ Start up

This program takes 2mins to start up, which is loading the page table and lexicon table to the main memory.

➢ Search

This program takes 1 or 2 second to respond search query.

6. User Instruction

➢ Development Environment

OS: OSX 10.10.5

JDK: 1.8

Sever: Tomcat 7

➢ Set Up Instruction

Follow the steps bellow to run the program in your local laptop:

Step1: load the project to your IDE (I use IntelliJ for my own), the project is a JavaEE project using Tomcat as the sever.

Step2: Add the missing dependencies jar (which your IDE will notify you). These dependencies didn't come along as I didn't use Maven in the JavaEE project.

Step3 (optional): Install MongoDB on your local laptop if you don't have MongoDB.

Step4: Run the CrawlParser under index.parser to store the raw page contents and page urls to your local MongoDB and create index for them.

Step5: Run the Tomcat sever, wait the localhost: 8080 launch automatically. Enjoy the search journey!

7. Future Work

➢ Use AWS EC2 or AWS Elastic Beanstalk to run the program.

We run the program on the local laptop for now which has two disadvantages: 1) It is not easy for others to use this program 2) It takes a large local memory, and a large CPU to run the program especially in the indexing part.

- Improve the search speed.

  The search speed could be a little bit slower when the search query contains very popular words, and one possible place to improve is the nextGEQ() function, it could be better to keep a current block instead of loading and compress the same block every time .